Securing Web Service Compositions: Formalizing Authorization policies using Event Calculus

Mohsen Rouached and Claude Godart

LORIA-INRIA-UMR 7503
BP 239, F-54506 Vandœuvre-les-Nancy Cedex, France
{mohsen.rouached, claude.godart}@loria.fr

Abstract. Service composition is a fundamental technique for developing Web services based applications. As autonomous services are invoked through protocols, issues such as security must be taken into account. Thus, ensuring security in such a system is challenging and not supported by most of the security frameworks proposed in current literature.
This paper presents a formal model for composing security policies dynamically to cope with changes in requirements or occurrences of events. We address one particular issue - that of authorization within a Web services composition. In particular, we propose a dynamic authorization model which allows for complex authorization policies whilst ensuring trust and privacy between the components services.

1 Introduction

Service Oriented Computing (SOC) is gaining prominence as the technology of choice for integrating applications in diverse and heterogeneous distributed environments. It is widely recognized that one of the barriers preventing widespread adoption of this technology is a lack of products that support non-functional features of applications, such as security, transactionality and reliability. Such properties are of utmost importance for Web service composition languages to keep their promises. Security is a challeging aspect of Web service composition that has not been so far deeply investigated despite its importance [2, 3]. For instance, a first challenge is the definition, the verification, and the enforcement of security policies as the complexity of composite Web services grows. To cope with this complexity, it is useful to design a conceptual model that gives a structured way to think about security policies. Another challenge is that non-functional concerns should be addressed by external specifications for a better separation of concerns and for more modular composition specification. For example, if we extend WSBPEL with new constructs for each non-functional concern of the composition, it would evolve into a very complex language, which in turn would limit its acceptance. Furthermore, mixing the specification of the core logic of the composition with specifications of security features and other non-functional concerns into one unit would make the composition specification too complex and hard to maintain and evolve.

In this paper, we propose to use a formalism based on the Event Calculus ($\mathcal{EC}$) [5] to specify authorization policies for Web services implied in Web services compositions. $\mathcal{EC}$ is interesting because it supports the direct representation of events that are used in such policies, and the advantage of such a formalism is that it allows for having a common representation for different security models, every service having its own security model. Given this common representation, we use it for two aspects. The first one deals with consistency checking. It means that when we have all the policies expressed in the $\mathcal{EC}$, we are able to check the consistency of the Web services composition with respect to security requirements. The second aspect is dedicated to conformance checking of the composition. Based on our previous works [7], this means that we are able to monitor and to detect security violations during the composition execution.

In the rest of the paper, we introduce in Section 2 the notion of authorization in the context of Web services composition. In Section 3, we present how we specify the policies using the $\mathcal{EC}$, and how the consistency can be checked. Section 4 is dedicated to related works. Finally, Section 5 concludes the paper and outlines some future directions.

2 Managing Authorization for Composite Web Services

Service Oriented Architecture allows for considerably more complex interaction models than the classical client/server model, including symmetric peer-to-peer interactions where both parties want to check authorisations, or multi-party composed services where authorization is an issue for each component service. Therefore, an appropriate authorization framework is needed to smooth the flow of a transaction between multiple services whilst respecting the privacy of the data used. This is a complex task since each individual service may have its own authorization requirements. The traditional authorization service – a third party can be used to enforce privacy between the two parties – is not appropriate in this kind of interactions where a coordinating service would need to exchange policy and credential information as well as managing the operation details. Managing these authorization exchanges can lead to processing bottlenecks within the service as well as privacy concerns given that the coordinating service retains visibility and control. In this context, our interest is about dynamically composed services.

A statically composed service would consist of a number of potentially independent services, each service having its own set of policies that must be satisfied for any particular request. A service invocation must satisfy the overall composed policy in order to proceed successfully. This kind of composition suggests a simple way to extend authorization for composite services where there is a well-defined combination of services and associated policies. However, the situation can be far more complex as a transaction is formulated in a more dynamic manner. Services can be added as needed (on demand) and may be derived from a dynamic service registry or through an auction. The user may want each service to meet appropriate policies. In these cases, authorization can be seen as a dialog. For

example, it is likely that a client interacts with a travel agent who in turn interacts with various service providers. In these situations, there is an interaction as the service provider pulls together a transaction over multiple components. To perform the transaction involving all these parties, the client must satisfy the union of all the individual policies. Once the service provider knows each subpart of the transaction is satisfied, it can allow the client to commit to the overall transaction.

In the rest of the paper, we focus on the later type of service composition since it is more realistic and needs a high level of details.

3 Formalizing Authorization for Composite Web Services

3.1 Basic notations and definitions

Ensuring security is tricky and formal notations are increasingly used to specify security policies. In this work, we use two booleans $autho^+$ and $autho^-$ to model positive and negative authorizations respectively. Therefore a positive authorization is denoted by $autho^+(s, o, a)$, where s, o, and a stand for subject, object, and action respectively. This authorization holds if the value of $autho^+(s, o, a)$ equals true and does not hold otherwise. Similarly, $autho^-(s, o, a)$ models a negative authorization. Positive and negative authorizations are used at the specification level to state who is or is not allowed to do what. As we will show, the use of signed (i.e positive/negative) authorizations gives more flexibility in handling authorization rules. For example, negation can be banned in consequences of rules without lost of generality.

An authorization policy must determine at any time the access rights of each subject with respect to any object and any action. Writing a complete specification to state this can be very complex and cumbersome. Indeed, to enable the definition of authorization policies in the context of Web service compositions, we assume a finite set of users U, Web services S, and roles R. Users (that can be services) are entities connecting to the system and allowed to submit requests. Web services are those involved in the composition process. Roles are named collection of privileges needed to perform specific activities in the service composition process. Hierarchical relationships can be defined within any of these sets to describe dependencies among their elements. For example services can be organized into federations. Each Web service can play several roles in the same composition, and can belong to more than one composition at the same time. In this case it will be able to exercise the union of the privileges of all these roles. How this union is to be determined and whether the user looses hidher personal privileges upon activation of some roles depend on the access decision policy to be applied. Note that the fact that the authorizations given to a role are applicable only when that role is active for a user has two advantages. First, it gives the user all those privileges that are needed to perform a task. Second, it is consistent with the "principle of least privilege": each process is confined to those actions needed to perform the task. This acts as a defense against malicious attacks that may aim to exploit the role's authorizations.

Formally, we define the set of roles related to a Web service composition C as R_C, and the role assigned to a given Web service s in that composition as r_C^s. Then, we distinguish between two states of a given Web service. It depends on its role in the request. The first one is given by s_{src} to express that the service s represent the source (who submit the request). The second type is denoted by s_{targ} to precise that the service s is the target (who receive the request).

To summarize, a service is seen as a resource that is provided within the system, to which access is controlled. A service can also request other services and is actively involved in computation. In our formal policy model, a Web service can therefore be seen as both source and target. The type of request made to the Web service is modelled as an action.

3.2 Authorization Model

To allow the necessary level of control over the behaviour of the Web service composition it is our contention that authorization policies should be defined in a language flexible enough to allow the specification of conditions that can include multiple triggering events that may take place over time. $\mathcal{EC}$ seems to be the best basis to start from.

We adapt a simple classical logic form of the EC, whose ontology consists of (i) a set of time-points isomorphic to the non-negative integers, (ii) a set of time-varying properties called fluents, and (iii) a set of event types (or actions). The logic is correspondingly sorted, and includes the predicates $Happens$, $Initiates$, $Terminates$ and $HoldsAt$, as well as some auxiliary predicates defined in terms of these. $Happens(a, t)$ indicates that event (or action) a actually occurs at time-point t. $Initiates(a, f, t)$ (resp. $Terminates(a, f, t)$) means that if event a were to occur at t it would cause fluent f to be $true$ (resp. $false$) immediately afterwards. $HoldsAt(f, t)$ indicates that fluent f is true at t. The auxiliary predicate $Clipped(t1, f, t2)$ expresses whether a fluent f was terminated during a time interval $[t1, t2]$. Similarly, the auxiliary predicate $Declipped(t1, f, t2)$ expresses if a fluent f was initiated during a time interval $[t1, t2]$.

To achieve a complete specification that supports formal reasoning in $\mathcal{EC}$, the following elements must be represented in the model.

- Separation between source services (s_{src}) and target services (s_{targ}) depending on the role of the service when performing or receiving the effect of an operation.
- Functions that can be used as parameters in the basic predicate symbols of $\mathcal{EC}$. We define these functions as events that may occur during the composition execution. Below, the introduced events are explained. In these formulas, V_p represents the set of parameters values for the operations supported by services.
  - $operation(s, Action(V_p))$: used to denote the operations specified in a policy function or event (see below).
  - $requestAction(s_{src}, operation(s_{targ}, Action(V_p)))$: represents the event that occurs whenever a service source attempts to perform an operation

on a target service. Therefore, this is the event that will trigger a permission (or denial) decision to be taken by the target service's access controller.

- $doAction(s_{src}, operation(s_{targ}, Action(V_p)))$: represents the event of the action specified in the operation term being performed by the service s_{src} on the service s_{targ}.
- $rejectAction(s_{src}, operation(s_{targ}, Action(V_p)))$: the event that occurs after the enforcement decision to reject the request by a particular source service to perform an action is taken.
- $permit(s_{src}, operation(s_{targ}, Action(V_p)))$: represents the permission granted to a source service to perform the action defined in the operation on the target service.
- $deny(s_{src}, operation(s_{targ}, Action(V_p)))$: used to denote that the source service, s_{src}, is denied permission to perform that action on the target service s_{targ}.

– In addition to the described $\mathcal{EC}$ predicates, we add specific predicate symbols. Indeed, in our case many of the function definitions above contain the tuple $(s_{src}, operation(s_{targ}, Action(V_p))$. To check if the members of this tuple are consistent with the specification of the Web service composition, we define the $isValidComp$ predicate. As such it must be used in any rule where functions with the tuple $(s_{src}, operation(s_{targ}, Action(V_p))$ are involved.

Having specified these elements, it is now possible to explain how the various symbols defined above can be incorporated into rules that represent the different types of information required to specify authorization policies able to support Web service composition requirements in terms of security.

The complete authorization enforcement model is illustrated in Figure 1. As shown, once the service source makes a request to perform an action on the service target, the target service's access controller processes it. To do this, the access controller evaluates the request by referring to the policy repository and the access control model. If the action is permitted, the access control model will proceed to do the requested action. Otherwise, if the action should be denied, the access control system will reject the action. We precise that the scheme is symmetric, i.e each of the two services could be target, source, or target and source at the same time.

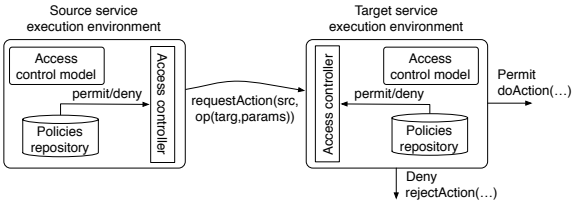

Fig. 1. Authorization Enforcement Model

As shown in Figure 1, we distinguish two scenarios to represent the enforcement model. The first scenario models the behaviour of the target service's access

controller, generating a *doAction* event when an action is permitted. This event would trigger the relevant service behaviour rules thus causing the composition state to change according to the specification. The second one models a target service's access control monitor rejecting the action to prevent a denied operation from being performed.

3.3 Authorization Specification

In order to correctly interact with the enforcement model described above, each policy specification rule should initiate the appropriate policy function symbol (permit, deny) for each of the events. So for example, a positive authorization policy rule should specify that $permit(s_{src}, Operation(s_{targ}, Action(V_p)))$ holds when the $requestAction(s_{src}, Operation(s_{targ}, Action(V_p)))$ event occurs and the constraints that control the applicability of the policy hold. Additionally, the fluent $permit(s_{src}, Operation(s_{targ}, Action(V_p)))$ should cease to hold once the action has been performed thus making it possible to re-evaluate the policy rule on subsequent requests to perform the action. The $\mathcal{EC}$ representation of this functionality is indicated in the $auto^+$ specification shown in Figure 2. This also shows how each of the other policy types would be represented by rules in the formal notation. For each rule, the terms, s_{src}, s_{targ}, $Action$ and $Constraint$, can be directly mapped to the source service, target service, action, *constraint* and event clauses used when specifying policies. The $Constraint$ predicate is introduced to specify the pre- and post-conditions for each operation. It can be represented by a combination of $HoldsAt$ terms.

The $autho^-$ specification shown in Figure 2 represents a negative authorization policy by stating that, if the $Constraint$ holds and the event requesting the action is performed happens, the action is denied. The second part of the rule shows how the *deny* fluent will be terminated once the decision to reject that action has been taken, thus allowing the specification to be re-evaluated on subsequent requests. Note that the termination parts for these policies do not have any constraints and can be generically specified for the whole service composition.

3.4 Conflicts

Using the authorizations' specifications presented in Figure 2, a Web service composer must be able to perform the two following functionalities: (1) given the policies of services to be involved in the composition, it must find if it exists a combination that satisfies each service policy to answer a given request, (2) given the composition of a set of services having their own policies, it must be able to prove that this composition is consistent regarding the policy of each service.

In order to detect conflicts involving authorization policies, i.e. those that arise when it exists two policies defined for the same source, target and action: one being an authorization and the other one being a prohibition, we introduce the $authConflict$ predicate that holds if an authorization conflict is detected.

Policy	Specification
$autho^{+}$	$Initiates(requestAction(s_{src}, operation(s_{targ}, Action(V_p))), permit(s_{src}, operation(s_{targ}, Action(V_p))), t1) \leftarrow isValidComp(s_{src}, operation(s_{targ}, Action(V_p))) \wedge Constraint$
	$Terminates(doAction(s_{src}, operation(s_{targ}, Action(V_p))), permit(s_{src}, operation(s_{targ}, Action(V_p))), t1) \leftarrow isValidComp(s_{src}, operation(s_{targ}, Action(V_p)))$
$autho^{-}$	$Initiates(requestAction(s_{src}, operation(s_{targ}, Action(V_p))), deny(s_{src}, operation(s_{targ}, Action(V_p))), t1) \leftarrow isValidComp(s_{src}, operation(s_{targ}, Action(V_p))) \wedge Constraint$
	$Terminates(rejectAction(s_{src}, operation(s_{targ}, Action(V_p))), deny(s_{src}, operation(s_{targ}, Action(V_p))), t1) \leftarrow isValidComp(s_{src}, operation(s_{targ}, Action(V_p)))$

Fig. 2. Event Calculus Specification for Authorization Policies

This predicate is defined as:

$HoldsAt(authConflict(s_{src}, operation(s_{targ}, Action(V_p))), t1) \leftarrow$
$\quad HoldsAt(permit(s_{src}, operation(s_{targ}, Action(V_p))), t1) \wedge$
$\quad HoldsAt(deny(s_{src}, operation(s_{targ}, Action(V_p))), t1)$

Let consider a typical example of authorization conflict, which arises when the same service is assigned to two roles that have opposite authorization permissions. To enable a complete specification of the different conflict cases that may arise, we introduce a further set of predicates, events, and fluents.

The additional predicates are $Service(name)$, $Action(name)$, $Role(name)$, and $ContradictoryRoles(r1, r2, t, a)$. $Service(name)$ denotes a service with a name *name*. $Action(name)$ defines an action with a name *name* that a source can process on a target. $Role(name)$ determines a role with the name *name*. $ContradictoryRoles(r1, r2, t, a)$ describes that roles $r1$ and $r2$ have opposite permissions for processing an action a at t.

Then, the events introduced are $AssignServiceRole(s, r)$ that denotes a request of a service s for assignment to a role r, $RolePermitAction(r, a)$ that specifies a request for permission of an action a for a role r, and $RoleDenyAction(r, a)$ that defines a request for denial of action a for a role r.

Finally, three fluents are specified: $Assigned(s, r)$ indicates that service s is assigned to a role r, $RoleHavePermission(r, a)$ defines that a role r is permitted to process action a, and $AuthorizationConflict(r1, r2)$ denotes that there is an authorization conflict in the composition (a service is assigned to contradictory roles).

Considering the elements described above, it is possible to define rules that can be used to recognise conflicting situations in the authorization policy specification. These rules are formalized as shown in Figure 3.

The first rule initiates the fluent $RoleHavePermission(r, a)$ when the event $RolePermitAction(r, a)$ happens if this fluent is currently not true. The second rule implements deny for role r to process the action a as a termination of fluent $RoleHavePermission(r, a)$ when $RoleDenyActivity(r, a)$ event happens. The third rule assigns service s to the role r when $AssignUserRole(s, r)$ event

Rule	Specification
R1	$Initiates(RoleHavePermission(r,a), RolePermitAction(r,a), t) \leftarrow$ $Happens(RolePermitAction(r,a), t) \wedge (\neg HoldsAt(RoleHavePermission(r,a), t))$
R2	$Terminates(RoleHavePermission(r,a), RoleDenyActivity(r,a), t) \leftarrow$ $Happens(RoleDenyActivity(r,a), t) \wedge HoldsAt(RoleHavePermission(r,a), t)$
R3	$Initiates(Assigned(s,r1), AssignUserRole(s,r1), t) \leftarrow$ $Happens(AssignUserRole(s,r1), t) \wedge (\neg HoldsAt(AuthorizationConflict(r1, r2), t))$
R4	$ContradictoryRoles(r1,r2,t,a) \leftarrow (HoldsAt(RoleHavePermission(r1,a), t) \wedge (\neg HoldsAt(RoleHavePermission(r2,a), t)))\mid(HoldsAt(RoleHavePermission (r2,a), t) \wedge (\neg HoldsAt(RoleHavePermission(r1,a), t)))$
R5	$Happens(conflictEvent, t) \wedge Initiates(AuthorizationConflict(r1,r2),$ $conflictEvent, t) \leftarrow HoldsAt(Authorized(s,r2), t) \wedge Happens(Authorize - Request(r1,s), t) \wedge ContradictoryRoles(r1,r2,a,t)$

Fig. 3. Rules for Authorization Conflicts

happens if $AuthorizationConflict(r1, r2)$ between the role $r1$ and some other role $r2$ is not presented in the composition process. The fourth rule defines two roles, one of which has and another one does not have permission for some action. Here we note that we not fix which role has positive permission and which role has negative permission. Thus, *ContradictoryRoles* is symmetrical regarding $r1$ and $r2$. Finally, the fifth rule defines a notion of authorization conflict: the user requested the assignment for the second of two contradictory roles.

4 Related Work

There are few papers on security in the context of Web service compositions. We are aware only of the work presented in [4], which presents an access control framework for business processes in BPEL. The theoretical framework identifies an interactive access control model based on logical abduction as a way for protecting security interests of the process and its partners. Like our's, this framework is specific to the authorization problem. Our proposal is more formalized and it can be easily applicable to more security facets in Web service compositions (confidentiality, integrity, and authentication). In [8] the authors present a tool giving a simplified, business-policy-oriented view to its users, who are configuring secure Web services in their systems. They also based their proposal on WS-Security and WS-Policy but their tool does not support composite Web services.

In the project SECTINO[1], a system architecture for local and global workflow system is proposed based on the XACML[6] and SAML. Security concerns are defined in OCL(Object Constraint Language) with model-driven UML tools. XACML is good for specifying policy in a specified domain. But it is not semantic rich enough for cross-organisational orchestration and high-level security requirements.

[1] http://qe-informatik.uibk.ac.at

AO4BPEL[1] proposes an aspect-oriented extension to BPEL. It uses aspects-oriented concept to modularize cross-cutting concerns like security and performance in business processes. Although the AO4BPEL framework offers the modularity and dynamic adaptability to the Web service composition, it lacks semantic description of security aspects, business processes and business rules. This make conflicts detection and policy negotiation infeasible for securing the Web service composition.

5 Conclusion

In this paper, we presented a framework for managing authorization policies for Web service compositions. Specifically, we have described the use of Event Calculus and abductive reasoning for developing a language that supports specification and analysis of authorization policies for Web service composition. A complete implementation and an EC plug-in for Web service were developed and tested using test cases.

There are several directions for future work to further improve the presented work. One thread in our future work will focus on the generalisation of the reasoning technique to handle other security properties as presented in Section **??**. A more general direction considers the application of our framework to other non-functional concerns in Web service compositions such as reliability, persistence, and transactional perspectives.

References

1. A. Charfi and M. Mezini. Aspect-oriented web service composition with ao4bpel. In *ECOWS*, volume 3250 of *LNCS*, pages 168–182. Springer, 2004.
2. D. Geer. Taking steps to secure web services. *IEEE Computer*, 36(10):14–16, 2003.
3. P. Hung, E. Ferrari, and B. Carminati. Towards standardized web services privacy technologies. In *Proc of the IEEE International Conference on Web Services (ICWS'04)*, San Diego, CA, USA, July 2004.
4. H. Koshutanski and F. Massacci. An access control framework for business processes for web services. In *XMLSEC '03: Proceedings of the 2003 ACM workshop on XML security*, pages 15–24, New York, NY, USA, 2003. ACM Press.
5. R. Kowalski and M. J. Sergot. A logic-based calculus of events. *New generation Computing 4(1)*, pages 67–95, 1986.
6. T. Moses. Extensible access control markup language (xacml) version 2.0 3, Feb 2005.
7. M. Rouached, O. Perrin, and C. Godart. A contract-based approach for monitoring collaborative web services using commitments in the event calculus. In *Sixth International Conference on Web Information System Engineering (WISE05)*, pages 426–434, 2005.
8. M. Tatsubori, T. Imamura, and Y. Nakamura. Best-practice patterns and tool support for configuring secure web services messaging. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, page 244, Washington, DC, USA, 2004. IEEE Computer Society.